Declarative XML Data Cleaning with XClean

Melanie Weis[1] and Ioana Manolescu[2]

[1] HPI für Softwaresystemtechnik GmbH
Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam
melanie.weis@hpi.uni-potsdam.de
[2] INRIA Futurs
2-4, rue Jacques Monod, 91893 Orsay Cedex France
Ioana.Manolescu@inria.fr

Abstract. Data cleaning is the process of correcting anomalies in a data source, that may for instance be due to typographical errors, or duplicate representations of an entity. It is a crucial task in customer relationship management, data mining, and data integration. With the growing amount of XML data, approaches to effectively and efficiently clean XML are needed, an issue not addressed by existing data cleaning systems that mostly specialize on relational data.
We present XClean, a data cleaning framework specifically geared towards cleaning XML data. XClean's approach is based on a set of cleaning operators, whose semantics is well-defined in terms of XML algebraic operators. Users may specify cleaning programs by combining operators by means of a declarative XClean/PL program, which is then compiled into XQuery. We describe XClean's operators, language, and compilation approach, and validate its effectiveness through a series of case studies.

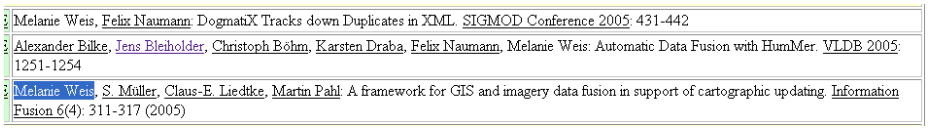
1 Motivation

Data cleaning is the process of correcting anomalies in a data source, that may for instance be due to typographical errors, formatting differences, or duplicate representations of an entity. It is a crucial task in customer relationship management, data mining, and data integration. Relational data cleaning is performed in specialized frameworks [14, 21, 26], or by specialized modules in modern relational database management systems [8].

With the growing popularity of XML and the large volumes of XML data becoming available, approaches to effectively and efficiently clean XML data are needed. For example, consider DBLP[3] whose data is available in XML format. Fig. 1 shows an excerpt of the DBLP entry of one of this paper's authors, on which we observe several XML data cleaning issues. First, the SIGMOD conference is represented by the conference abbreviation, the string "Conference", and the year of the conference, whereas VLDB is only represented by its abbreviation and year. Clearly, both conferences are represented differently, which can be corrected through data cleaning. A second example is the representation of author names. In the bottom publication, the first author is represented by its firstname and lastname, whereas the second author's firstname is

[3] http://www.informatik.uni-trier.de/ ley/db/

abbreviated. The last inconsistency is that the bottom publication has actually not been written by the same author as the other two publications. When looking at the paper, the first author is represented as M. Weis, and it has been falsely matched to author Melanie Weis. This problem, known as entity resolution [4] is also part of data cleaning. This example shows that XML data cleaning is a problem of practical relevance. Therefore, we develop XClean, a system for declarative XML data cleaning.


Melanie Weis, Felix Naumann: DogmatiX Tracks down Duplicates in XML. SIGMOD Conference 2005: 431-442

Alexander Bilke, Jens Bleiholder, Christoph Böhm, Karsten Draba, Felix Naumann, Melanie Weis: Automatic Data Fusion with HumMer. VLDB 2005: 1251-1254

Melanie Weis, S. Müller, Claus-E. Liedtke, Martin Pahl: A framework for GIS and imagery data fusion in support of cartographic updating. Information Fusion 6(4): 311-317 (2005)

Fig. 1. Excerpt of DBLP entry

In developing such a system, lessons learned from relational data cleaning clearly apply, but have to be rethought due to the significantly different data structure.

Modularity. Data cleaning processes should be *modular* in order to allow the composition of such processes from a set of smaller, interchangeable building blocks. Modularity brings several benefits. It facilitates reusing existing cleaning transformations, simplifies the process of debugging and inspecting the data transformation process, and it allows incremental development, maintenance and evolution of the cleaning process. To achieve modularity, relational data cleaning systems such as [14] have defined cleaning operators. For XClean, we also define operators, which distinguish themselves from existing relational cleaning operators because they have to deal with the nested and semi-structured nature of XML data. For example, object properties may be multi-valued (e.g., a publication has several authors) or missing (opposed to empty content). Furthermore, crucial information describing the way XML nodes relate to one another is given by their parent-child relationships, whereas relational data cleaning concentrates on cleaning flat tuples of a single table. Consequently, XML data cleaning operators need to preserve these relationships, but also have the opportunity to exploit them.

DBMS-backed data cleaning. Many transformations involved in data cleaning are closely related to those typically applied inside database management systems (DBMSs). Therefore, cleaning data on top of a DBMS allows taking advantage of its functionalities, including persistence, transactions etc. but also query optimization, which may speed up the cleaning process. Relational data cleaning's reliance on RDBMSs was limited by expressive power mismatches between the cleaning primitives and SQL. Features such as user-defined aggregate functions, transitive closure computation, nested tables etc. are either not fully supported by the language, or not well supported by existing systems. In contrast, the standard XML query language, XQuery, is Turing-complete, raising the question whether simply writing XQuery queries may not suffice for data cleaning? While this approach can be made to work, it amounts to writing fresh code for every new cleaning problem, which does not agree with the last requirement applying to relational data cleaning systems, as well as to XClean.

Declarativity. By declaratively describing the cleaning process, its logic can be decoupled from the actual processing and its implementation. This makes data cleaning processes easier to write and to debug than alternative approaches, based on imperative code. Declarative cleaning programs allow concentrating on the cleaning tasks, while delegating storage and optimization issues to the underlying data management

systems. As XML cleaning operators are significantly different from relational cleaning operators, it is natural that declaring them is also different. In XClean, we provide a declarative programming language, called XClean/PL to specify the cleaning process. This program is then compiled to an XQuery, and executed using any XQuery engine.

In this paper, we present XClean, the first modular, declarative system for native XML data cleaning. Our main contributions are (i) The definition of *cleaning operators*, to be combined in arbitrary complex cleaning processes, viewed as operator graphs. (ii) A high-level *operator definition language*, called XClean/PL, which is *compiled* into XQuery, to be executed on top of any XQuery processor.

We outline the XClean architecture and define its operators, in Sec. 2. XClean/PL and its compilation to XQuery are outlined in Sec. 3. We evaluate XClean's expressive power and ease of use on several case studies in Sec. 4, discuss related works in Sec. 5, then we conclude. Due to the lack of space, we only briefly cover the different aspects of the system, and refer interested readers to the project website [1] for further details.

2 XClean Overview

The XClean system described in this paper is a data cleaning system that allows declarative and modular specification of a cleaning process. In this section, we first present the overall XClean system, and then introduce the XClean operators enabling modularity.

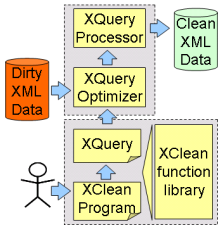
2.1 XClean Architecture

The architecture of the XClean system is depicted in Fig. 2(a). A user specifies an XClean program in our proposed declarative XClean/PL language (see Sec. 3). An XClean/PL program specifies a set of XClean operators, and the way their inputs and outputs are connected. This program is then compiled into the XQuery standard language [29], whose query plan can be optimized before it is executed. The result of the XQuery is the clean XML data. Let us now discuss the different components.

Function library. XClean provides a function library including commonly used functions (e.g., date formatting for scrubbing, edit distance for string similarity) which may be used in XClean/PL programs. The function library can be easily extended by users, as they can simply add XQuery functions to the library, which can be implemented either in XQuery or in an external language [29].

XClean/PL Program We design XClean/PL with the goal of minimizing the cognitive effort for the average XClean user. XClean/PL provides a custom syntax for cleaning-specific operators, increasing the *readability and ease of maintenance* of cleaning programs, while being significantly more *concise* than the resulting XQuery programs.

XQuery Compilation, optimization, and execution. XQuery is a feature-rich language widely implemented by major DBMS vendors (such as IBM, Oracle, Microsoft etc.), and free-source projects (e.g. Saxon, BerkeleyDB/XML etc.), so using XQuery allows to execute the programs on top of any XQuery-enabled platform. XQuery execution plans can be optimized to make query execution more efficient, a task achieved by XQuery optimizers, implemented by several XQuery engine providers. Executing the XQuery results int the clean XML data. XQuery optimization is a separate research area by itself; the study of particular optimization issues raised by XClean/PL translated programs is part of our future work. In this paper, we focus on XClean operators, XClean/PL, and its compilation to XQuery.



(a) XClean architecture

Operator	Goal
Candidate Selection (CS)	Select elements to be cleaned.
Scrubbing (SC)	Remove errors in text (typos, format, ...).
Enrichment (EN)	Specify data that supports cleaning.
Duplicate filtering (DF)	Filter non-duplicate element pairs.
Pairwise duplicate classification (DD)	Classify pairs of elements as duplicates, non-duplicates, ...
Duplicate clustering (DC)	Determine clusters of duplicates.
Fusion (FU)	Create unique representation of an entity.
XML view (XV)	Create XML view of clean data.

(b) XClean operator averview

Fig. 2. Overview of the XClean architecture and its cleaning operators.

Our choice of XQuery as the target language for XClean/PL compilation is pragmatic, driven by the wide current availability of XQuery processors. However, the read-only XQuery requires cleaning to disassemble data fragments to be cleaned, and then to re-assemble them again in the clean result, as will become clear when discussing the process in more detail. The process may thus not be very performant. The possibility to clean data *by updating it* is thus very interesting, and is provided, e.g., by the W3C XQuery Updates [28] W3C draft, and by more recent programming-style XQuery extensions [7]. As current XQuery processors start supporting such languages, they may become XClean/PL compilation targets, providing both the advantages of query optimization (for read-only cleaning parts) and those of "in-place" cleaning via updates.

2.2 Operators

XClean's cleaning operators are summarized in Fig. 2(b), and are defined over a data model described in detail in [16], for which we only summarize features and notations relevant for defining XClean operators' algebra.

Any XClean operator inputs and outputs collections of (nested) tuples, having the structure $(\$a_1 = val_1, \ldots, \$a_k = val_k)$, where each $\$a_i = val_i$ is a variable-value pair. Variable names such as $\$a_1$, $\$a_2$ etc. are \$-prefixed, following XQuery conventions, and are unique within a tuple. A value may be (*i*) the special constant $\perp$ (*null*), (*ii*) an XML node or value, or (*iii*) a (nested) set, list, or bag of tuples. Given a tuple $(\$a_1 = val_1, \ldots, \$a_k = val_k)$ the list of names $[\$a_1, \ldots, \$a_k]$ is the tuple's *schema*.

We refer to the set of all tuples as $\mathcal{T}$, and denote the set of n-ary tuples $\mathcal{T}_n$. We use $\mathcal{P}(\mathcal{T})$ to denote all sets of tuples from $\mathcal{T}$. Given a tuple $t = (\ldots \$x = v \ldots)$ we say that $\$x$ maps to v in the context of t. We represent by $t.\$x$ the value that the variable $\$x$ maps to in the tuple t. The notation $t' = t + (\$var = v)$ indicates that the tuple t' contains all the variable-value pairs of t and, in addition, the variable-value pair $\$var = v$. The tuple $t'' = t + t'$ contains all the variable-value pairs of both t and t'.

Fig. 3 (bottom) presents a sample XML document containing three versions of the same real-world movie, with their respective title, year and actor sets. The labels $m1$, $a1$ etc. uniquely identify an element and are used to reference them in our example. Assume that the goal of the cleaning process is: (i) obtaining one representation for each movie, including *all alternative titles*, *one year*, and *all actors (but each actor only once)*, and (ii) *restructuring each actor element* into a firstname and a lastname element. A possible result of this process is shown at the top of Fig. 3. Using this example, we introduce the XClean operators that define the cleaning process. The process and its intermediary results can be followed from the bottom up on Fig. 3.

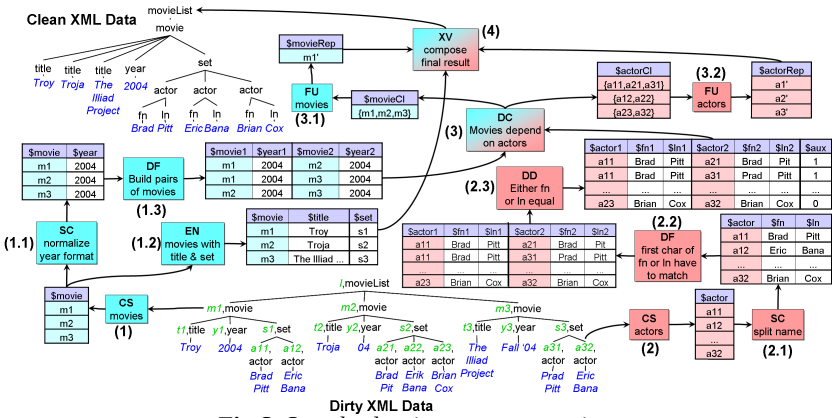


Fig. 3. Sample cleaning process overview.

Candidate Selection Candidate selection is used to designate elements that are subject to the cleaning process. Candidates are designated by a set of queries $q_1, q_2, \ldots, q_k$, and the effect of the CS operator is to evaluate all queries and union their results into a flow of 1-tuples. Candidate selection is the first step in the process of cleaning, therefore, the CS operator has no input (child) operator. Formally:

$$CS_{q_1,q_2,\ldots,q_k}() = q_1 \cup q_2 \cup \ldots \cup q_k$$

Consider the selection of movie candidates. Let $q_m =$ $doc/moviedb/movie. Fig. 3 depicts the operator CS_{q_m} (1) and its output. Similarly, actor candidates are selected by the operator CS_{q_a} (2) in Fig. 3, where $q_a =$ $doc/moviedb/movie/set/actor.

Scrubbing Scrubbing is used for normalizing and standardizing formats and/or values. We model this based on a set of *scrubbing functions*, which apply on (tuples of) atomic values and produce (tuples of) atomic values. For generality, XClean scrubbing functions may have one or several inputs and one or several outputs. We deliberately chose to restrict scrubbing functions to atomic values. We argue that functions which apply more complex object analysis and transformation for cleaning would benefit from being decomposed in elementary steps, which help reasoning and optimization. Our framework does allow to model such transformations by the XML View operator, described later. Formally, let $f : \mathcal{A}^n \rightarrow \mathcal{A}^m$ be a scrubbing function, and IN be a flow of tuples of arity k, such that $n \leq k$, and let $i_1, i_2, \ldots, i_n$ be a set of integers such that for any $1 \leq j \leq n$, we have $1 \leq i_j \leq k$. Furthermore, let $q_1, q_2, \ldots, q_n$ be some XML queries, which are used to extract from the (potentially complex) input the atomic inputs of the scrubbing function. Then:

$$SC_{f,i_1,i_2,\ldots,i_n,q_1,q_2,\ldots,q_n}(IN) = \{t + (\$b_1 = v_1^t, \$b_2 = v_2^t, \ldots, \$b_m = v_m^t) \mid t \in IN, \exists (u_1^t, u_2^t, \ldots, u_n^t) \; s.t. \; u_1^t \in q_1(t.\$a_{i_1}), u_2^t \in q_2(t.\$a_{i_2}), \ldots, u_n^t \in q_n(t.\$a_{i_n}), \text{ and } f(u_1^t, u_2^t, \ldots, u_n^t) = (v_1^t, v_2^t, \ldots, v_m^t)\}$$

where $\$b_1, \$b_2, \ldots, \$b_m$ do not appear in IN's schema. The definition accounts for the general case where each XML query q_i may return a sequence of results. SC semantics requires that every combination of atomic inputs be used to call f.

To split actor names into a firstname and a lastname, we use a scrubbing function $f_{actor} : \mathcal{A} \rightarrow \mathcal{A}^2$. Let $q_{actor} =$ ./actors/actor/text() be the query extracting the initial

actor names, and assume IN contains tuples having just one attribute, namely the actor candidates. We apply $SC_{f_{actor},1,q_{actor}}(IN)$ (2.1) in Fig. 3. The second scrubbing operation is the standardization of years, performed by applying $SC_{f_{year},1,q_{year}}(IN)$ (1.2): $f_{year} : \mathcal{A} \rightarrow \mathcal{A}$ scrubs movie year values (by normalizing them to four digits), and $q_{year} =$ ./year/text(). This time, IN contains the set of movie candidates.

Enrichment Enrichment allows specifying which data to use for comparing two candidate duplicates. Let IN be a flow of tuples, $\$c$ be the name of one attribute in these tuples, and $q_1, q_2, \ldots, q_k$ be a set of XML queries, which may be absolute (i.e., navigate from the root of some given document) or relative (i.e., navigate from $\$c$). We have:

$$EN_{\$c,q_1,\ldots,q_k}(IN) = \{t + (\$a_1 = q_1(t.\$c), \ldots, \$a_k = q_k(t.\$c)) \mid t \in IN\}$$

where $\$a_1, \$a_2, \ldots, \$a_k$ do not appear in IN's schema.

Consider a movie is described by its title and its set. Therefore, we specify $q_{en1} =$ ./title, and $q_{en2} =$ ./set. Assuming as input the result of the movie candidate selection operator CS_{q_m}, the operator $EN_{\$movie,q_{en1},q_{en2}}(IN)$ corresponds to step (1.2).

Duplicate Filtering Duplicate filtering constructs (a subset of) the cartesian product of a flow of candidates with itself, to be used in order to identify duplicate objects later. If only a subset of the cartesian product is built, the operator has been used to restrict the space of comparisons by pruning out some pairs of objects about which it can be said with certainty that they are not duplicates. Only on the pairs of objects which may be duplicates, other measures will later be applied to determine whether they are duplicates indeed. The second, separate output of this operator is the set of pairs of input tuples, which are definitely deemed to be non-duplicates. Although they will not be used in the main cleaning process, they may be needed, e.g., for further analysis by the user. Formally, let m be the arity of the tuples in the input IN. Let $f_1, f_2, \ldots, f_k$ be k functions such that $f_i : \mathcal{T}_m \times \mathcal{T}_m \rightarrow \{true, false\}$. The duplicate filtering operator DF has two outputs, denoted DUP and $NODUP$, and is defined as follows:

$$DF_{f_1,f_2,\ldots,f_k}.DUP(IN) = \\ \{t_1 + t_2 \parallel t_1 \in IN, t_2 \in IN, f_1(t_1,t_2) = f_2(t_1,t_2) = \ldots = f_k(t_1,t_2) = true\}$$
$$DF_{f_1,f_2,\ldots,f_k}.NODUP(IN) = \\ \{t_1 + t_2 \parallel t_1 \in IN, t_2 \in IN, \exists 1 \leq i \leq k \ s.t. \ f_i(t_1,t_2) = false\}$$

Clearly, several DF operators can be used to apply (conjunctively) several filters on potential duplicates. More complex (not necessarily conjunctive) filtering combinations can be devised by creating a complex function from simple ones, and using a single DF operator based on the complex function.

Let $f_{firstLetter}$ be a filter function that returns true if the string values of either $firstname or $lastname of an actor are equal, false otherwise. Further, let f_{equal} return true if the $actor nodes are equal according to node identity, false otherwise. Finally, let f_{order} return true if the $actor node of the first tuple appears before the $actor node of the second tuple in the document. Then, $DF_{f_{firstLetter},f_{equal},f_{order}}$ is the operator labeled (2.2) in Fig. 3, and its DUP output is the table depicted right above it.

Pairwise Duplicate Detection Duplicate detection expects input tuples that include two possibly enriched candidates, and outputs one or more tuple classes, according to a classifier function. If only one class of output is produced, it is understood as containing duplicates. If more classes are produced, their semantics depend on the classifier. For instance, one classifier may identify "certain duplicates", "likely duplicates" on which another duplicate detection classifier on pairs is applied, and "others", for which human user expertise is needed. Moreover, each tuple is annotated with a classifier-produced data structure which may encapsulate auxiliary information about the classification result (such as the confidence in the announced score, similarity, etc.).

Formally, let $f_{class} : \mathcal{T}_{2m} \rightarrow \{1, 2, \ldots, m\} \times \mathcal{N}$ be a classifier function returning for every input tuple, the index of a class, and an auxiliary data structure, modeled as an XML node. The duplicate detection operator DD has m outputs, denoted $OUT_1, OUT_2, \ldots, OUT_m$, defined as follows:

$$DD_{f_{class}}.OUT_i(IN) = \{t + (\$n = f_{class}(t).aux) \mid t \in IN, f_{class}(t).c = i\}$$

where the class output of the classifier is denoted $f_{class}.c$, the auxiliary data structure is denoted $f_{class}.aux$, and $\$n$ does not appear in IN's schema.

We detect duplicates in actors using a classifier $dActor$ returning a single DUP class. It classifies a pair of actors as duplicates if either firstname or lastname are equal (this simple function could already have been used for filtering, but we use it here to keep the example simple). The auxiliary information of the classifier returns the edit distance between the names. The operator DD_{dActor} is numbered (2.3) in Fig. 3.

Duplicate Clustering Duplicate clustering takes as input one or several sets of potential duplicates, and outputs as many sets of duplicate clusters. A tuple in every output has one attribute, whose value is a set of tuples from the corresponding input flow, representing a set of candidates which represent the same real-world object. Clustering algorithms need to examine their whole input before producing their output, therefore, this operator is not defined on a per-tuple basis, as the previous ones. Moreover, some clustering algorithms take advantage of candidates from one input to determine how to cluster candidates from another input [22, 11], therefore this operator has multiple inputs and outputs. Formally, let k be an integer, and $IN_1, IN_2, \ldots, IN_k$ be some operators such that tuples in the output of IN_i, $1 \leq i \leq k$, have arity $2\,n_i$, for some integer n_i. $\{IN_i\}$ denote the set of tuples output by IN_i. Let

$$f_{clust} : \mathcal{P}(\mathcal{T}_{2\,n_1}) \times \mathcal{P}(\mathcal{T}_{2\,n_2}) \times \ldots \times \mathcal{P}(\mathcal{T}_{2\,n_k}) \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{T}_{n_1})) \times \mathcal{P}(\mathcal{P}(\mathcal{T}_{n_2})) \times \ldots \times \mathcal{P}(\mathcal{P}(\mathcal{T}_{n_k}))$$

be a clustering function that takes as input k whole sets of tuples, and outputs k sets of sets of tuples (representing clusters). Let $\{IN_i\}$ denote the set of tuples output by IN_i, and $OUT_1, OUT_2, \ldots, OUT_k$ be the outputs of DC. Then:

$$DC_{f_{clust}}(IN_1, IN_2, \ldots, IN_k).OUT_i = f_{clust}(\{IN_1\}, \{IN_2\}, \ldots, \{IN_k\}).i$$

where the i-th attribute of f_{clust}'s output is denoted OUT_i. Note that DC breaks down every two-candidate duplicate in two, folding all duplicate tuples into a single cluster.

Consider the clustering of movies, described by their titles and actor sets. To detect duplicates in movies, information about duplicates among their actor sets is helpful, so we perform duplicate detection using clustering on the input IN_{actors}, consisting

of the pairs produced by actor duplicate detection (2.3), and IN_{movies}, holding movie pairs. In our example, the clustering operator $DC_{f_{clust}}(IN_{actors}, IN_{movies})$ is labeled (3), and it produces two sets of tuples. Each tuple in the first set is a cluster of actors considered duplicates, and each tuple in the second set is a cluster of duplicate movies.

Fusion The fusion operator applies on clustered tuples. Its purpose is to construct a single representative, or cleaned version, from every cluster of tuples in its input. Formally, let $f_{fuse} : \mathcal{P}(\mathcal{T}) \rightarrow \mathcal{T}$ be a function that, for every cluster of $\mathcal{T}$ tuples, returns a cleaned tuple representing the unified cluster. Assume IN contains 1-tuple attributes, such that every attribute value belongs to $\mathcal{P}(\mathcal{T})$. Then:

$$F_{f_{fuse}}(IN) = \{f_{fuse}(t.\$a_1) \mid t \in IN\}$$

This generalizes to IN having several attributes, one of which is a nested table.

We fuse movies by unifying all their descendant sequences, which results in a new element denoted m'_i, for the i-th tuple in the input (3.1). We fuse actors by choosing the first actor (according to document order) as a cluster representative (3.2).

As for duplicate detection, fusion may involve more complex logic than the simple aggregation above. E.g., when detecting duplicates in movies, and simultaneously in actors, which are descendants of movies, the fused result of movies also depends on the fusion of actors. Hence, information about duplicate movies and actors is required, similarly as for relationship-based duplicate detection. As fusion usually requires a previous clustering, we decide to let fusion be part of the clustering function when necessary.

XML View During cleaning, it may be necessary at several points to apply some "adjustment" transformation to one operator's output prior to sending it into another operator's input. Furthermore, if only parts of the input data have been cleaned, an extra query may be needed to combine the cleaned data with the document it originated from. Such transformations can be accomplished via the XV operator, standing for XML View. Let IN contain some tuples in $\mathcal{T}_n$, and $i_1, i_2, \ldots, i_k$ be some column indices in IN. Let $q(\$x_1, \$x_2, \ldots, \$x_k)$ be a parameterized XML query. Then:

$$XV_{q,i_1,i_2,\ldots,i_k}(IN) = \{t + (\$a = q(t.\$a_{i_1}, t.\$a_{i_2}, \ldots, t.\$a_{i_k})) \mid t \in IN\}$$

Actor elements need to be restructured in the final result, with ⟨firstname⟩ and ⟨lastname⟩ children. Names have been split by the scrubbing operator, we now need to create a new representation of every candidate movie, including the complex-structure actor names. This transformation can be specified by an XQuery q_{xv}. Fig. 3 shows the $XV_{q_{xv}}(IN)$ operator (4) and its output.

3 XClean Programming

Having discussed the XClean architecture and the operators used to define a cleaning process, we present how these can be specified using the programming language XClean/PL, and compiled to an executable XQuery.

3.1 Language Rationale and Design

The specification of a cleaning process can be decomposed in (i) choosing the specific filters, distance functions, duplicate detection algorithms, etc., and (ii) writing the "surrounding" code necessary to implement the operator tree using these functions.

Previous experience in data cleaning [14, 21, 26] demonstrates that creating or choosing the cleaning functions and algorithms requires a human expert, and cannot be automated. In contrast, the second task is repetitive, and amenable to automation. Based on this observation, we designed the XClean/PL language as follows.

An XClean/PL program is a set of clauses, each of which defines a cleaning operator. Operators input and output tuples from shared, global XClean/PL variables. Sample XClean/PL clauses appear in Tab. 1(a), the full description of XClean/PL's syntax being provided in [1]. XClean/PL keywords appear in bold font.

The top enrichment clause defines the operator labeled (1.2) in Fig. 3. The clause refers to two named tuple sets, globally visible in the XClean/PL program: $scrubbedMovies, the operator's input, and $enrichedMovies, its output. The tuple variable $m iterates over the input. The **BY** clause introduces the two enrichments: the result of each query is added as a new variable, part of the output flow.

The cluster classification clause defines the operator labeled (3) in Fig. 3. The classifier function xcl:radc denotes a relationship-aware duplicate clustering function (e.g.,[22]), which is one among the possible classifiers to be used here. The classification function returns two sets of clusters, one containing movies and another one actors. The **INTO** keyword is used, as previously, to capture the outputs of xcl:radc, and make them visible in the XClean program for further usage. This clause also explicitly renames the attributes in each set of cluster's schema, through the **SCHEMA** clause.

The last clause in Tab. 1(a) defines the pairwise duplicate operator DD_{dActor} used for illustration in Section 2.2 (numbered 2.3 in Fig. 3). Function detectActorDups returns 1 if the two actors are considered duplicates and 2 otherwise, while function eDist is a simple edit distance. The **WITH** keyword is immediately followed by a call to the classifier function, after which the (optional) function producing the auxiliary information is invoked. The difference between the WITH keyword and the USING keyword is that the first calls a function within an iteration, whereas the second calls a function that takes sets of tuples as input.

3.2 Compiling XClean/PL to XQuery

An XClean/PL program is compiled based on a few principles, discussed next. Tab. 1(b) shows XQuery snippets obtained when compiling the XClean/PL clauses of Tab. 1(a).

First, given that XQuery does not support tuples, every tuple manipulated during the cleaning process is translated into a system-generated XML element, named ⟨tSYS⟩. For every variable of an XClean tuple, the system-generated ⟨tSYS⟩ element has a child element named after the variable (without the leading $), its content being variable value. Nested tuples are translated into nested ⟨tSYS⟩ elements. While this element generation is generally computationally expensive, [12] has characterized situations when element construction can be avoided. These are the situations when the *identity* of the constructed node is never used in the remainder of the same XQuery program. Fortunately, all ⟨tSYS⟩ element creation done in XClean are in this situation, thus, optimized XQuery processors may support this quite well.

Second, global XClean/PL variables are compiled in XQuery variables introduced by let clauses, bound to the lists of ⟨tSYS⟩ elements.

Third, XClean/PL operators defined by iterating over input tuples (scrubbing, enrichment, duplicate filtering, fusion, and XML view) are compiled into for-where-return

(a) XClean/PL clauses	(b) Compiled XQuery clauses
ENRICH $m **IN** $scrubbedMovies **INTO** $enrichedMovies **BY** $m.mCand/movie/title/text() **AS** $title, $m.mCand/movie/set/actor **AS** $set;	let $enrichedMovies := for $m in $scrubbedMovies return <tSYS>{$m/element(), element title {$m/mCand/movie/title/text()}, element set {$m/mCand/movie/set/actor} </tSYS>
CLUSTER CLASSIFICATION USING xcl:radc($actorDups, $candMovieDups) **INTO** $movieClusters **SCHEMA** [$movieCluster], $actorClusters **SCHEMA** [$actorCluster];	CLUSTERSSYS := xcl:radd($actorDups, $candMovieDups), $movieClusters := for $VSYS in $CLUSTERSSYS/element()[1]/element() return <tSYS>{element mClust{$VSYS/element()}}</tSYS>, $actorClusters := for $VSYS in $CLUSTERSSYS/element()[2]/element() return <tSYS>{element aClust{$VSYS/element()}}</tSYS>
PAIR CLASSIFICATION **PAIR** $p **IN** $candActorDups **WITH** detectActorDups($p), eDist($p) **AS** $aux **INTO** $actorDups **IF CLASS** = 1;	$PAIRSSYS := for $p in $candActorDups return <CLASSSYS>{ attribute CLASSSYS{detectActorDups($p)}, <tSYS>{$p/element(), element aux{eDist($p)}}</tSYS> }</CLASSSYS>, $actorDups := $PAIRSSYS[@CLASSSYS = '1']/element()

Table 1. Sample XClean/PL clauses.

expressions, with XClean/PL's iteration variables (such as $m) compiled into XQuery for clause variables.

Finally, XClean adds an internal xcl:id identifier attributes to XML elements manipulated during cleaning. One reason for this is related to XQuery semantics: when creating an element having the value of the variable $x as a child, such as, e.g., ⟨a⟩{$x}⟨/a⟩, the nodes associated to $x are *copied*, thus they are no longer equal to the original nodes. However, the cleaning process needs to reason on the relationships between the nodes, e.g., when de-duplicating movie candidates based on related actor duplicates. Moreover, when re-assembling the cleaned elements (last step in Fig. 3), IDs are also needed. A second usage of system-introduced IDs is to enable *lineage tracing*, i.e., discovering the operators (and inputs) that have led to obtaining a given output (clean) element. Lineage issues are often central in data cleaning processes [14, 26], to help users understand cleaning results, inspect and refine the process. To keep ID insertion and manipulation overhead low, XClean IDs are added to cleaning candidates only.

4 Usage report

The approach described in this paper has been implemented in our XClean Java-based prototype ([23]) following the architecture shown in Fig. 2(a). XClean/PL programs are compiled using the antlr tool[4], into XQuery programs.

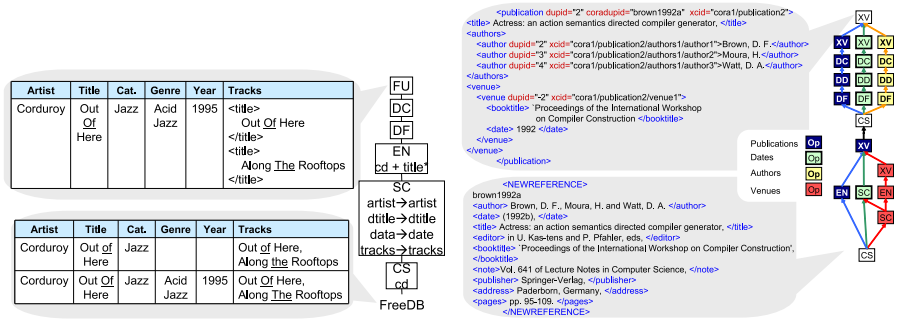
Section 4.1 reports on the usage of XClean to devise several data cleaning processes of different complexity and scope, focusing on XClean's expressive power and on the benefits of modular and declarative XML data cleaning. Section 4.2 outlines a quantitative evaluation of XClean performance. Details on use cases, sample data sets, full XClean/PL programs, and their resulting XQueries are available at [1].

4.1 Use Cases

FreeDB Use Case. This use case concerns CD description FreeDB data[5]. The cleaning process (see Fig. 4(a)) includes correcting errors in artist names (common errors include different capitalization schemes, Various Artists is also represented by V.A.,

[4] http://www.antlr.org/

[5] http://www.freedb.org



(a) CDDB Use Case (b) CORA Use Case

Fig. 4. CDDB and CORA use case description

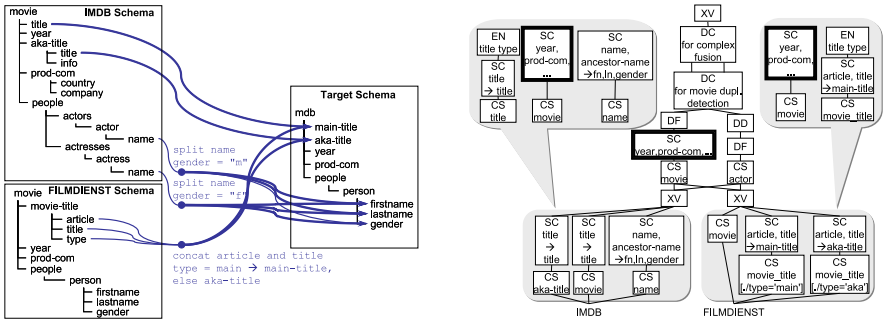
Various), standardizing dates, correcting titles (e.g., the title element often includes Artist/Title), and track titles (again, capitalization). All these operations correspond to scrubbing operators. We further enrich CDs with track ⟨title⟩ elements obtained by splitting the comma-separated list of track titles into individual elements. Using the enriched CDs, the final task is to deduplicate CDs: if both ⟨artist⟩ and ⟨title⟩ are equal, we consider CDs to be duplicates, which is a boolean function without auxiliary information that can directly be used in the DF operator. Clustering performs transitive closure over CD pairs and fusion creates a single representative for every CD. During fusion, conflicts may appear in category, genre, year, and tracks: different categories, genres, and years are concatenated, whereas sets of ⟨title⟩ elements are unified. Note that the table representation has only been used for readability, the actual data is XML.

MOVIE Use Case. This is a data integration scenario, in which movies from two sources are first mapped to a common schema, and then de-duplicated. The sources are the Internet Movie Database IMDB and the German Movie Repository FILMDIENST[6]. Fig. 5(a) outlines the two source schemas and the target schema. In IMDB titles, the possible leading "The" or "An" is separated in an ⟨article⟩ element. Non-trivial correspondences between source and target types are rendered by curved arrows, possibly annotated with transformation functions. For instance, IMDB names are split into firstname and lastname, and gender is set to "m" for actors, or to "f" for actresses.

XClean allows specifying this process in several ways, as illustrated in Fig. 5(b). In this figure, the central tree of connected operators represents one possible cleaning process, denoted P_1. An alternative XClean operator graph for the same task, which we denote P_2, can be obtained by modifying P_1, as we will shortly explain.

In P_1, the operators in the shaded bottom areas are used to select and scrub the cleaning candidates from IMDB (left), and FILMDIENST (right). From IMDB, we extract: aka-title, movie and name candidates. Then, title scrubbing (upper/lower case normalization) is applied on the text content of aka-title elements, as well as on the text of movie/title; actor name scrubbing separates first names from last names. FILMDIENST candidates are: movies, movie main titles, and movie aka-titles. Both title types undergo scrubbing. The XV operators align the movies to the target schema as shown in Fig. 5. The CS operators merge candidate movies (respectively, candidate actors) from both

[6] [7] and http://film-dienst.kim-info.de/, respectively



(a) Plan P_1 (b) Plan P_2

Fig. 5. Two equivalent cleaning processes for the MOVIE Use Case

data sources, to be used together in the rest of the process: year and prod-com scrubbing, duplicate filtering and clustering. Note that the second duplicate clustering operator performs fusion, illustrating a case of complex fusion that requires several inputs (movies and actors). The final result is composed using an XV operator.

The second plan P_2 differs from P_1 in the following two ways: First, the scrubbing operator on year and prod-com (thick line) is pushed down below the intermediate XV operators. Because integration has not been performed below this point, the scrubbing operator is split into two individual scrubbing operators, one for each source. Second, scrubbing in P_1 has been performed separately on main-titles and aka-titles in both sources. In P_2, titles of each source are scrubbed in one operation, are then enriched by their type and split in the corresponding title elements in the intermediate XV operator.

This scenario illustrates the benefits of *modular* data cleaning: the same scrubbing function can be used for all titles. It also demonstrates the interest of *declarativity*: a given process can be specified in multiple ways, and separating its specification from its actual implementation allows for its automatic optimization.

CORA Use Case. The CORA bibliographic data set is frequently used to evaluate duplicate detection algorithms [11, 22]. Fig. 4 outlines an XClean operator graph (right) as well as a sample input reference (bottom left) and its corresponding clean version (bottom right). Colors of operator boxes indicate the bibliographic data components on which they apply, e.g. authors, dates etc. The sample XClean process scrubs, enriches and restructures the dirty data (from the bottom to the upper CS). We assume the restructured data is then fed into three parallel cleaning chains, with the purpose of comparing their respective clean outputs.

In this example, we again scrub dates, reusing a standard function available in the XClean function library already used in the MOVIE scenario. Also, detecting duplicates in author names is similar to detecting duplicates in actor names, so we can reuse the same pairwise duplicate detection function as in the MOVIE scenario, showing the advantage of modularity.

4.2 Quantitative Aspects

We have run the XQuery programs on several freely available XML engines: XML Spy, QizX/Open, and Saxon B[8]. We found the latter to be the most efficient for our

[8] http://www.altova.com, http://www.xfra.net/qizxopen/, and http://saxon.sourceforge.net/

generated XQuery set. Tab. 2 provides an overview of the size of the data sets of the three use cases, actual runtimes (averaged on 10 runs), and savings in word counts when using XClean/PL, relative to the word count of the respective generated XQueries.

Use Case	#Nodes	#Chars	Time (s)	Word Savings
FreeDB	2001	36103	4.7	61%
MOVIE (P_1)	3108	9666	20.4	59%
MOVIE (P_2)	3108	9666	1.8	57%
CORA	1116	9705	6.1	45%

Table 2. Use Case Statistics

All data sizes are of moderate size, but larger sizes did not fit in main-memory using Saxon B when it came to pairwise duplicate detection. Nevertheless, we clearly observe a difference on runtime, depending on the cleaning plan. Indeed, the difference between the two plans for the MOVIE scenario is of an order of magnitude. This is mainly due to the higher number of function calls and the multiple joins performed in the intermediate XV operator of P_1. Some of these joins are avoided in P_2, because candidates have already been enriched with the information via the EN operators. This recalls the classical optimization consisting of pushing function calls under joins, if the function call results are not cached [10]. As XQuery optimizers grow more efficient and include such mainstream techniques, the performance of XClean programs translated to XQuery is likely to improve. (Admittedly, more efficient XQuery processors are there today - on the Saxon website, the commercial version of Saxon is said to be two orders of magnitude faster, on some queries, than the free version we used.)

In the CORA use case, although the data set is quite small, the runtime is worse than for FreeDB or MOVIE (P_2), because it applies more expensive operators on pairs (DF, DD, DC).

Finally, we observe significant savings in the size of a XClean/PL program over the generated XQuery, as shown in the last column of Tab. 2. This indicates that a specialized language like XClean/PL makes the specification of cleaning tasks more concise, thus, we believe, more convienent for the user.

5 Related Work

Due to the lack of space, we only briefly discuss selected related work. A survey on relational data cleaning is made in [20], and more recent approaches include AJAX [14] and Potter's Wheel [21]. XClean is conceptually close to AJAX by its operator-based approach. However, our operators consider the existence of more than one candidate type, which can be related to each other. Relationships between candidate types are maintained throughout an XClean process, and can be used by various algorithms, e.g., for duplicate detection or fusion. Another difference to AJAX is that the XML context lifts the expressive power barriers that confronted AJAX. In our context, advantages of a declarative, modular approach are: ease of specification and maintenance, and opportunities for optimization. AJAX moreover provided an exception handling mechanism, which we plan to consider as well in the future.

XClean is not meant to replace existing algorithms for specific cleaning tasks, such as clustering, distance computation etc. Instead, these approaches can be plugged in as physical implementations of specific operators, thus re-using existing results and

running code. For duplicate detection, numerous algorithms have been developed, for relational data [15, 17], XML/hierarchical data [2, 19, 24], and more complex graph data [11, 22, 25]; a survey is provided in [27]. For similarity joins, the computationally expensive part of duplicate detection, a relational operator has been proposed in [9]. Fusion has received less attention, and all work focuses on relational data. The authors of [6] propose an operator that extends SQL to support declarative fusion and implemented in the HumMer system [5], and we plan to develop a similar technique for XML data. Other solutions include TSIMMIS [18] relying on source preference in the context of data integration, and ConQuer[13] that filters inconsistencies out of query results.

XClean's internal model includes tuples [1], which have made it easy to model associations between objects. Existing works suggested a controlled inclusion of tuples in XQuery to facilitate analytic queries rich in group-by [3]. The difference is that we include tuples as XClean internals and compile in standard XQuery, whereas [3] add new syntactic constructs.

6 Conclusion

We presented XClean, a system for declarative XML data cleaning. Users of the system write an XClean/PL program that reflects the desired cleaning process and which is automatically compiled into an XQuery, that can be optimized and executed by an XQuery engine. The result of this query is a clean version of the data. We defined several operators that can be combined in a modular way to form a cleaning process, and for each of which an XClean/PL clause exists. Use case based studies show that using XClean/PL to define a cleaning process is more convenient than writing a custom XQuery, and operators can be easily reused.

However, efficiency for a given cleaning task depends on the actual cleaning plan. The performance attained by the XQuery processors used in our evaluation could clearly be improved; as part of our future work, we intend to investigate which (intra-engine, external to XClean) XQuery optimizations would most help for such queries. XClean extensions we envision in the short term are: a GUI to support the design of the cleaning process, and exception handling (also absent from XQuery !), which is very important since exceptions may arise from a variety of sources in a cleaning context, and they include valuable information for the user seeking to refine the process.

Acknowldgements. This research was partly funded by a "DAAD Doktorandenstipendium" scholarship.

References

1. XClean: A system for declarative XML data cleaning. http://www.informatik.huberlin.de/~mweis/xclean/xclean.html.
2. R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, 2002.
3. K. Beyer, D. D. Chamberlin, L. Colby, F. Ozcan, H. Pirahesh, and Y. Xu. Extending xquery for analytics. In *SIGMOD*, 2005.
4. I. Bhattacharya and L. Getoor. A latent dirichlet model for unsupervised entity resolution. In *SIAM Conference on Data Mining (SDM)*, Bethesda, MD, 2006.

5. A. Bilke, J. Bleiholder, C. Böhm, K. Draba, F. Naumann, and M. Weis. Automatic data fusion with HumMer. In *VLDB*, 2005.
6. J. Bleiholder and F. Naumann. Declarative data fusion - syntax, semantics, and implementation. In *ADBIS*, 2005.
7. D. Chamberlin, D. Florescu, D. Kossmann, J. Melton, and J. Robie. Programming with XQuery. In *XIMEP Workshop*, 2006.
8. S. Chaudhuri, K. Ganjam, V. Ganti, R. Kapoor, V. Narasayya, and T. Vassilakis. Data cleaning in Microsoft SQL server 2005. In *SIGMOD*, 2005.
9. S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.
10. S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *VLDB*, 1993.
11. X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, 2005.
12. D. Florescu and D. Kossmann. XML query processing. In *ICDE*, 2004.
13. A. Fuxman, E. Fazli, and R. J. Miller. ConQuer: Efficient management of inconsistent databases. In *SIGMOD*, 2005.
14. H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, 2001.
15. M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, May 1995.
16. I. Manolescu and Y. Papakonstantinou. An unified tuple-based algebra for XQuery. Technical report, 2005.
17. A. E. Monge and C. P. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *SIGMOD-1997 DMKD Workshop*, May 1997.
18. Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *VLDB*, 1996.
19. S. Puhlmann, M. Weis, and F. Naumann. XML duplicate detection using sorted neigborhoods. In *EDBT*, 2006.
20. E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin, Volume 23*, 2000.
21. V. Raman and J. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, 2001.
22. P. Singla and P. Domingos. Object identification with attribute-mediated dependences. In *Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, Porto, Portugal, 2005.
23. M. Weis and I. Manolescu. Xclean in action (4 page demo, to appear). In *CIDR*, 2007.
24. M. Weis and F. Naumann. DogmatiX tracks down duplicates in XML. In *SIGMOD*, 2005.
25. M. Weis and F. Naumann. Detecting duplicates in complex XML data(poster). In *ICDE*, 2006.
26. J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.
27. W. E. Winkler. Overview of record linkage and current research directions. Technical report, U. S. Bureau of the Census, 2006.
28. The XQuery Update Facility. http://www.w3.org/TR/xqupdate, 2006.
29. XQuery 1.0. http://www.w3.org/TR/XQuery, 2006.